Generation and Analysis of Transition Systems

Hossein Hojjat

June 7, 2009

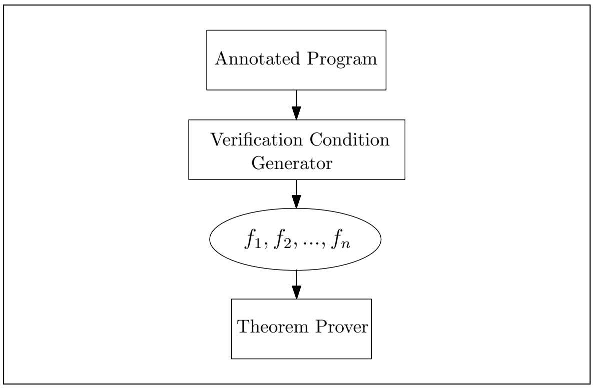

Figure 1: The overall process of VCG generaion

Abstract

In the common practice of co

1 Introduction

One of the main approaches in software formal verification is assertional reasoning based on Hoare logic [5]. This way a code is annotated in certain points with some assertions about the correctness of the program. An assertion guarantees that the program has the desired properties at the definition point From an annotated code a tool extracts a set of verification conditions (VC), which are then fed to a theorem prover for correctness proof. The big picture of the VC generation process is depicted in Figure 1.

The complex features such as pointers, memory allocation, data structures and concurrency can make VC generation challenging. In the program verifiers usually the original source code is first translated to a simple intermediate language. Deriving VC conditions from the simple intermediate language is easier. It is also useful when extending the verifier with more features, since the new capabilities may be translated to the intermediate language without the need to change the VC generator. The intermediate language is usually some flavor of the Dijkstra's guarded command language [3], and the VC conditions are generated using liberal precondition semantics. Some successful verifiers for the Java language include Krakatoa [6], ESC/JAVA [4] and Jahob [2]. Krakatoa translates JML-annotated Java programs to proof obligations for various interactive theorem provers. ESC/JAVA uses automated theorem proving for some particular classes of errors, and it is not sound. The main strength of Jahob in comparison to the others is its capability in reasoning about data structures.

In this project we suggest an intermediate language for the language Scala. Scala is a programming language being developed at the EPFL University. As a mixture of object oriented and functional paradigms, Scala allows a great deal of flexibility in programming. For a complete documentation for the language we refer to its webpage [1]. The proposed intermediate language uses the control flow graph (CFG) of a program. CFG shows all the paths that might be traversed through a program during its execution. We represent the graph using a logical formula. The corresponding formula of a CFG describes how the state of the program is changed during the transitions. A transition is described as a guarded command that executes only when the condition of the transition is satisfied. We use the Isabelle/HOL interactive theorem prover [7] to reason about the formula of a system.

An important merit of using a CFG as an intermediate language is it simplicity. CFGs are a common way to reason about the execution of most of the programs.

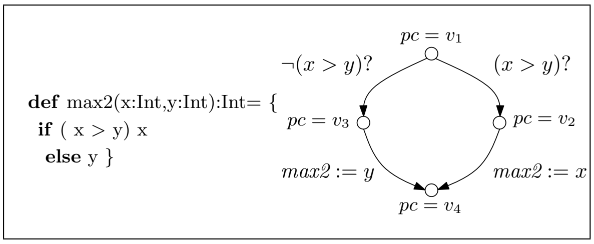

Figure 2: The CFG of a maximum function

Table 1: Mapping of arrays

Scala	Isabelle/HOL
xs : Array[Int]	xs :: "int list"
xs(5)	xs ! 5
xs.length	length xs
xs(1) = 4	list_update xs 1 4
xs map (x ⇒ x + 1)	map (λx. x + 1) xs

However, we are not sure know they can be compared with the previous approaches. We are in the process of implementing an automated compiler from Scala to a formula in Isabelle, and until then we cannot give a precise comparison.

1.1 Small Example

In this section we give a small motivating example which shows how are mapping works in general. Figure 2 shows both the code and CFG of a function that takes two integer arguments and returns their maximum value.

We exploit a similar approach to [8] in describing a transition system in Isabelle.

datatype *label* = $v1 \mid v2 \mid v3 \mid v4$
record *state* =
  pc :: label
  x :: int
  y :: int
  *max*2 :: int
definition *program* :: "(*state* × *state*) **set**"
where "*program* ≡{(s,s').
$(s' = s(\!|pc := v2|\!) \wedge (xs > ys) \wedge (pcs = v1))\vee$
$(s' = s(\!|pc := v3|\!) \wedge \neg(xs > ys) \wedge (pcs = v1))\vee$
$(s' = s(\!|pc := v4, max2 := xs|\!) \wedge (pcs = v2))\vee$
$(s' = s(\!|pc := v4, max2 := ys|\!) \wedge (pcs = v3))\}$"

Here s and s' denote the initial and final states, and the condition afterwards is the guard of the command. The variable *pc* controls the overall execution of the program in its path and prevents illegal jumping from one point to another point in the transition system. Let r be the relation representing the program and s be the relation representing the specification, program meets specification iff $r* \subseteq s$. Proving the correctness of the requirement in its general form is cumbersome and turns out to be difficult. We can test the program by unrolling the relation for some constant n, and then prove the weaker condition $r^n \subseteq s$. In the "max2" function since there are no loops and all paths of the program finish in two steps, we unroll the relation two times and show its correctness.

lemma *maximum* [*simp*]:
"(*program* ∘ *program* ∘ *program*) ⊆
$\{(s, s').(pcs = v1) \rightarrow (max2s' = max(xs)(ys))\}$"
apply (unfold program_def)
apply *auto*
done

In the rest of the report we describe how our mapping works for data structures, pattern matching, recursion and concurrency.

2 Data Structures

Many data structures can be represented with lists in Isabelle. As a simple example, consider the mapping of Scala arrays in Teble 1. In some cases the behavior of the Scala arrays may be different from its model in Isabelle. As an example the ArrayIndexOutOfBoundsException exception does not have a counterpart in our Isabelle model. We believe that these capabilities can be easily added to the translation. For the moment we do not consider these complications.

We apply the given mapping in the translation of a bubble sort (Figure 3) Scala code. The CFG of the code is shown in Figure 4.

The state of the program contains the following elements: pc :: label, xs :: "int list", res :: "int list", changed :: bool, a :: nat and tmp :: int. For example, the transition between labels $v6$ and $v7$ has the following form:

```
def bubblesort(xs:Array[Int]):Array[Int]={
var changed=false
do{
  changed=false
  for(a←0 until (xs.length - 1))
  if(xs(a)>xs(a+1)){
    var tmp=xs(a)
    xs(a)=xs(a+1)
    xs(a+1)=tmp
    changed=true}
  }while(changed)
xs
}
```

Figure 3: Bubble sort

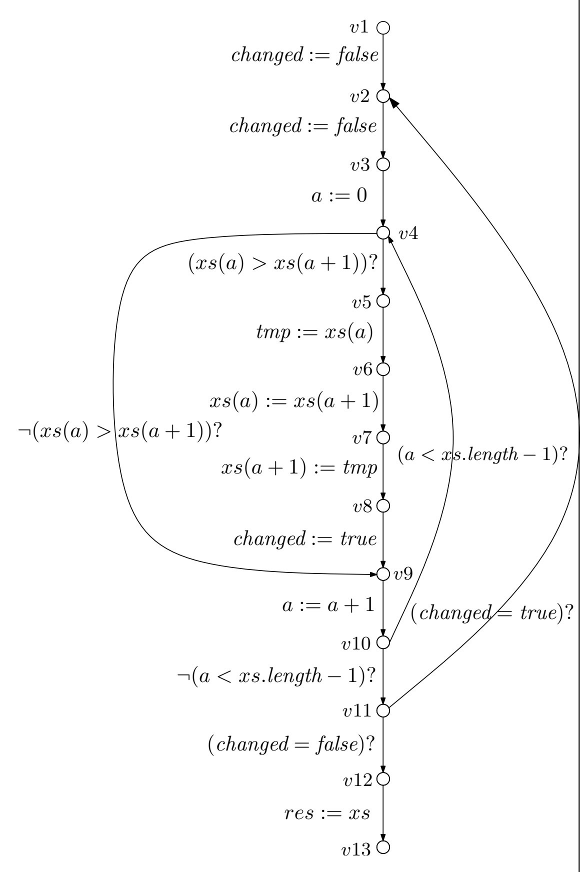

Figure 4: CFG of bubble sort

$(s' = s(\!|pc := v7,$
$xs := list\_update(xss)(as)((xss)!(as + 1))|\!)\wedge$
$(pcs = v6))$

Scala has a built-in general pattern matching mechanism. It allows to match on any sort of data with a first-match policy. For example, consider the following small example.

```
def matchTest(x: Int): String = x match {
  case 1 ⇒ "one"
  case 2 ⇒ "two"
}
```

The pattern matching cases are mapped to partial patterns in Isabelle. Since the guards are of type boolean, at the end of each partial matching we add $|\_ \Rightarrow False$. This makes the whole expression False whenever the pattern is not matched.

$((\textbf{case}\ (xs)\ \textbf{of}\ 1 \Rightarrow s' = s(\!|pc := v2, res := one|\!)$
$\wedge\ (pcs = v1)|\_ \Rightarrow False)\ \vee$
$(\textbf{case}\ (xs)\ \textbf{of}\ 2 \Rightarrow s' = s(\!|pc := v2, res := two|\!)$
$\wedge\ (pcs = v1)|\_ \Rightarrow False))$

3 Recursion

In a recursive computation some transitions of CFG are reused in each recursive call. To re-execute the transition we simply cannot loop back, since the local values of each call can be different from the previous ones. Similar to the method of compiling a recursive program, we make use of a stack in our translation. Stack is modelled with lists in Isabelle, the same as Section 2. Consider the following simple recursive factorial function:

```
def fact(n:Int):Int={
  if(n==0) 1
  else {
  val t=fact(n-1)
  n*t
  }
}
```

The CFG of the "factorial" function is depicted in Figure 5.

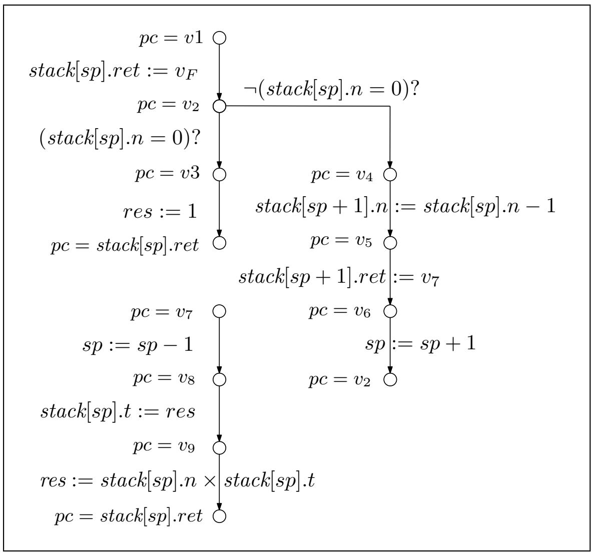

Figure 5: CFG of factorial

```
class Producer(c:Consumer) extends Actor{
   def act(){
      while(true) c!0
   }}
class Consumer extends Actor {
   var num = 0
   def act() {
   react{
   case msg ⇒ {
      num = num + 1
      act}
}}}
```

Figure 6: Producer-Consumer

4 Concurrency

The overall control of a program is preformed by the *pc* variable. We have to use a separate *pc*s for each concurrent process. Consider the producer-consumer code of Figure 6, and the corresponding CFG in Figure 6. Two different *pc* variables are used for modelling the probelm. The variable *pc*1 is used in producer and *pc*2 is used in consumer. If the number of concurrent processes are not known in advance, we consider an array of *pc* variables.

definition producer :: "(state × state) set"
where "$producer \equiv \{(s, s').\exists i.(s' = s(|$
$pc1 := list\_update(pc1 s)\ i\ v1,$
$msgbox := Cons0(msgbox s)|)$
$\wedge((pc1 s)!i = v1))\}$

If the process had also local variables then we used a stack to manage the different variables.

5 Future Work

We are implementing the proposed mapping for a subset of Scala. A bigger goal is to combine our ideas with the forthcoming theorem prover of the LARA group.

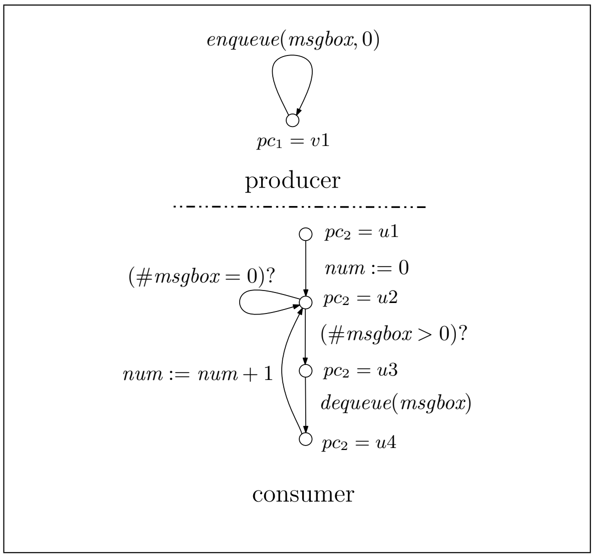

Figure 7: CFG of producer consumer

References

[1] `http://www.scala-lang.org/`

[2] C. Bouillaguet, V. Kuncak, T. Wies, K. Zee, M. Rinard, *Using First-Order Theorem Provers in the Jahob Data Structure Verification System*, Verification, Model Checking and Abstract Interpretation, 2007

[3] E. Dijkstra, *A Discipline of Programming*, Prentice Hall PTR, 1997.

[4] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, R. Stata, *Extended static checking for Java, SIGPLAN Not. Analysis of Object-Oriented Programs, v. 37, n. 5, pp. 234–245, 2002.*

[5] C. A. R. Hoare, An axiomatic basis for computer programming*, Communications of the ACM, v. 12, n. 10, pp. 576–580, 1969. pp. 12–32, 2008.*

[6] C. Marché, C. Paulin-Mohring, X. Urbain, The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML*, Journal of Logic and Algebraic Programming, v. 58, n. 1-2, pp. 89–106, 2004.*

[7] T. Nipkow, L. C. Paulson, M. Wenzel, Isabelle/HOL: a proof assistant for higher-order logic*, Springer-Verlag, 2002.*

[8] L. C. Paulson, Mechanizing UNITY in Isabelle*, ACM Transactions on Computational Logic (TOCL), v. 1, n. 1, pp. 3–32, 2000.*